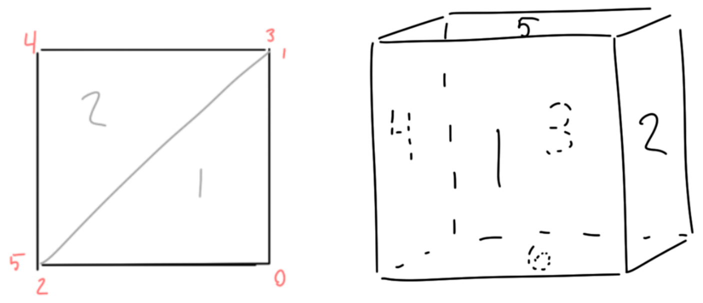
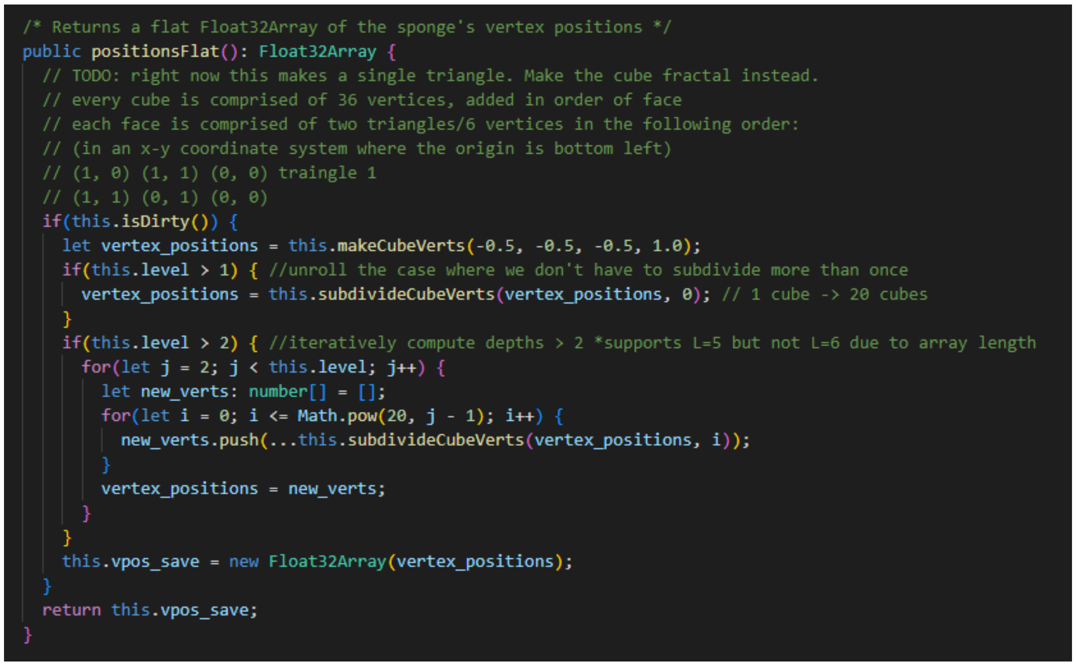
Matthew Jagen
mtj595

The Sponge:
I designed the sponge by thinking of it as a structure of cubes that could each be called on to split into 20 more cubes in the menger sponge structure. I didn't actually create a Cube class or struct, but keeping the data for each cube together in each of the vertex, index, and normal arrays helped the way I think about the project.

Each cube consists of 6 faces. Each face has 2 triangular faces and 6 vertices arranged in the data in this order, with the cube faces in this order:

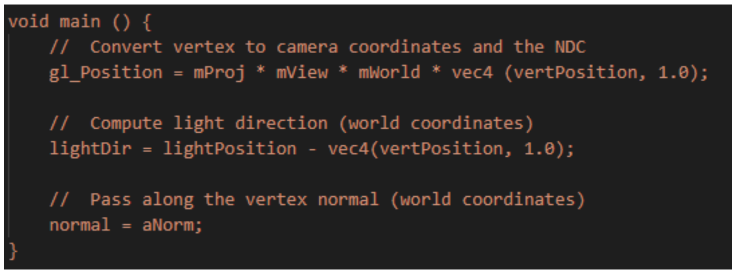


So, each cube has a total of 36 vertices/normals, and 12 triangular faces.
When the menger level is set to a number greater than 1, each cube is used to calculate the vertices of 27 sub cubes and then the 7 empty cubes are carved out. Because the normals are the same for every cube, I just calculate the number of final cubes based on the menger level and calculate the normal array all at once. The same process is used for the index array because the vertices are arranged in the vertex list in such a way that the index array is always 1, 2, 3, …, n.

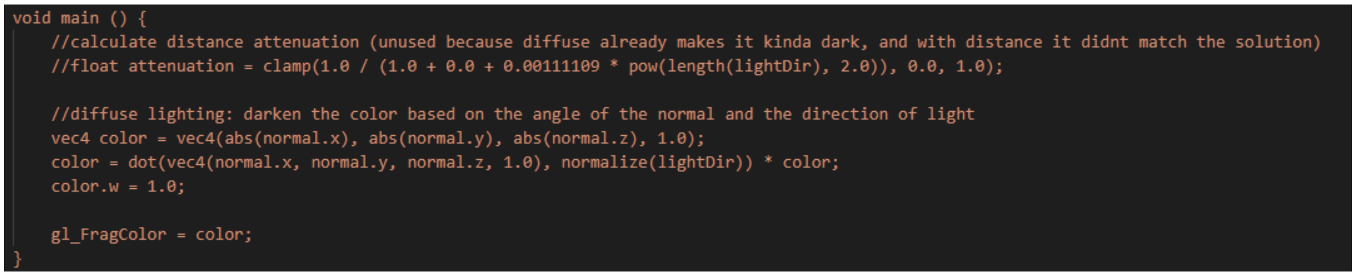


```
/* Returns a flat Float32Array of the sponge's vertex positions */
public positionsFlat(): Float32Array {
  // TODO: right now this makes a single triangle. Make the cube fractal instead.
  // every cube is comprised of 36 vertices, added in order of face
  // each face is comprised of two triangles/6 vertices in the following order:
  // (in an x-y coordinate system where the origin is bottom left)
  // (1, 0) (1, 1) (0, 0) traingle 1
  // (1, 1) (0, 1) (0, 0)
  if(this.isDirty()) {
    let vertex_positions = this.makeCubeVerts(-0.5, -0.5, -0.5, 1.0);
    if(this.level > 1) { //unroll the case where we don't have to subdivide more than once
      vertex_positions = this.subdivideCubeVerts(vertex_positions, 0); // 1 cube -> 20 cubes
    }
    if(this.level > 2) { //iteratively compute depths > 2 *supports L=5 but not L=6 due to array length
      for(let j = 2; j < this.level; j++) {
        let new_verts: number[] = [];
        for(let i = 0; i <= Math.pow(20, j - 1); i++) {
          new_verts.push(...this.subdivideCubeVerts(vertex_positions, i));
        }
        vertex_positions = new_verts;
      }
    }
    this.vpos_save = new Float32Array(vertex_positions);
  }
  return this.vpos_save;
}
```

The shaders for the sponge are very simple. The vertex shader converts the vertices from model space to NDC using the MVP and stores it in gl_position, as well as calculating the direction of the light to the vertex.
The fragment shader sets the color of the fragment to the normal (so that x is red, y is green, z is blue), and darkens it based on the angle between the normal and the direction of light.

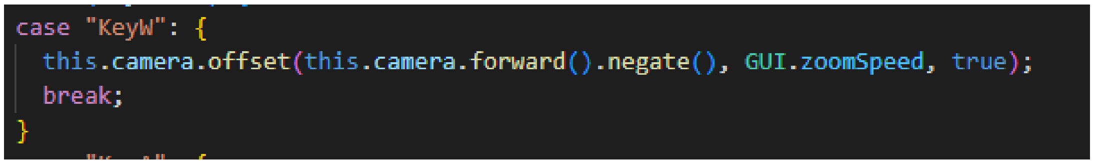


```
void main () {
    //  Convert vertex to camera coordinates and the NDC
    gl_Position = mProj * mView * mWorld * vec4 (vertPosition, 1.0);

    //  Compute light direction (world coordinates)
    lightDir = lightPosition - vec4(vertPosition, 1.0);

    //  Pass along the vertex normal (world coordinates)
    normal = aNorm;
}
```

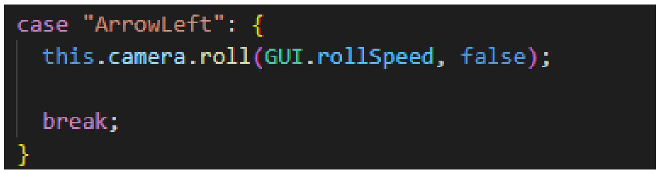


```
void main () {
    //calculate distance attenuation (unused because diffuse already makes it kinda dark, and with distance it didnt match the solution)
    //float attenuation = clamp(1.0 / (1.0 + 0.0 + 0.00111109 * pow(length(lightDir), 2.0)), 0.0, 1.0);

    //diffuse lighting: darken the color based on the angle of the normal and the direction of light
    vec4 color = vec4(abs(normal.x), abs(normal.y), abs(normal.z), 1.0);
    color = dot(vec4(normal.x, normal.y, normal.z, 1.0), normalize(lightDir)) * color;
    color.w = 1.0;

    gl_FragColor = color;
}
```

The Camera:
WASD and arrow key movement was very simple to implement. It just uses the Camera's built in offset() and roll() functions. Direction is taken from the Camera with forward(), right() and up().

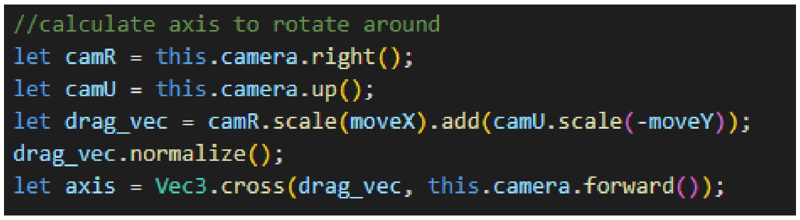


```
case "KeyW": {
  this.camera.offset(this.camera.forward().negate(), GUI.zoomSpeed, true);
  break;
}
```



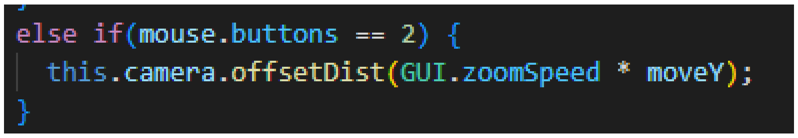
```
case "ArrowLeft": {
  this.camera.roll(GUI.rollSpeed, false);

  break;
}
```

Mouse dragging was a bit trickier. Each time that the drag event is called, it uses the previous X and Y position of the mouse on the screen to calculate the distance the mouse moved since the last call. It uses the drag distance then to calculate a drag direction vector in camera space and then rotates about the cross product between the drag vector and the camera's forward vector. This works out so that if the drag is directly upwards, the axis of rotation is directly left/right and sort-of 'spins' with the drag vector when the drag direction is diagonal.



```
//calculate axis to rotate around
let camR = this.camera.right();
let camU = this.camera.up();
let drag_vec = camR.scale(moveX).add(camU.scale(-moveY));
drag_vec.normalize();
let axis = Vec3.cross(drag_vec, this.camera.forward());
```

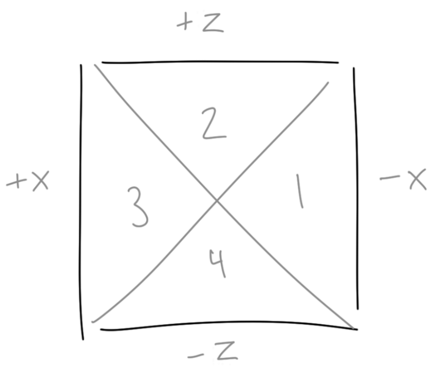
Zoom is much simpler in comparison, simply checking if the right mouse button is clicked or not.



```
else if(mouse.buttons == 2) {
    this.camera.offsetDist(GUI.zoomSpeed * moveY);
}
```

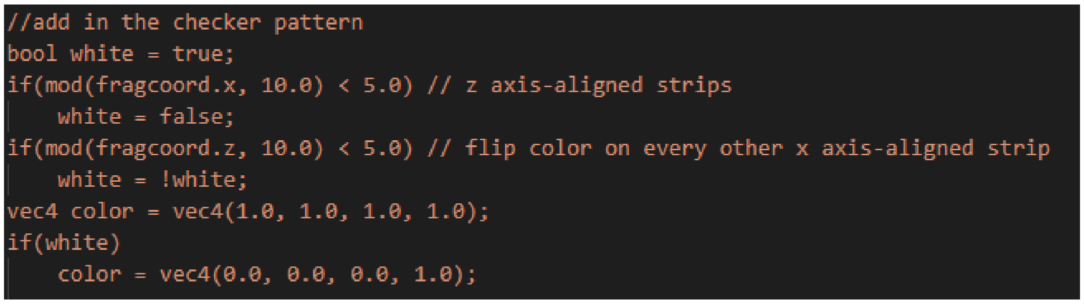
The Floor:

The floor is structurally similar to the sponge, except it only has 1 face of 4 triangles.



Instead of mapping the floor to have model coordinates of (-1, y, -1) to (1, y, 1) and doing some trickery with the projection matrix, I just set the corner values to be really high for the illusion of an infinite floor.

The checkerboard effect is implemented in the fragment shader. The vertex shader passes the vertex's position in world coordinates to the fragment shader, and then each fragment's color is determined by the world coordinates of that fragment (a vec4 called fragcoord).

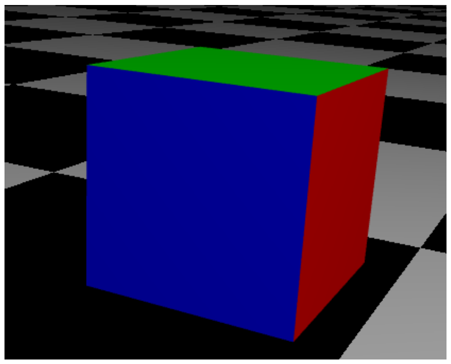
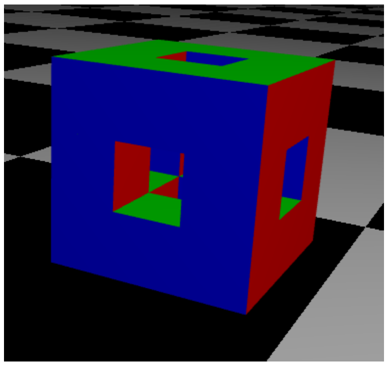
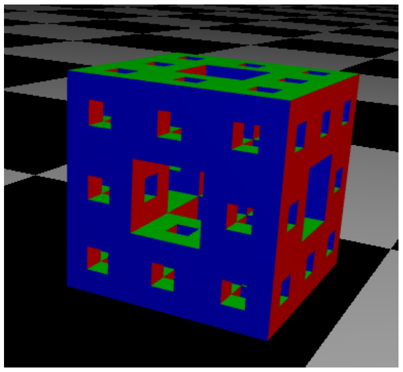
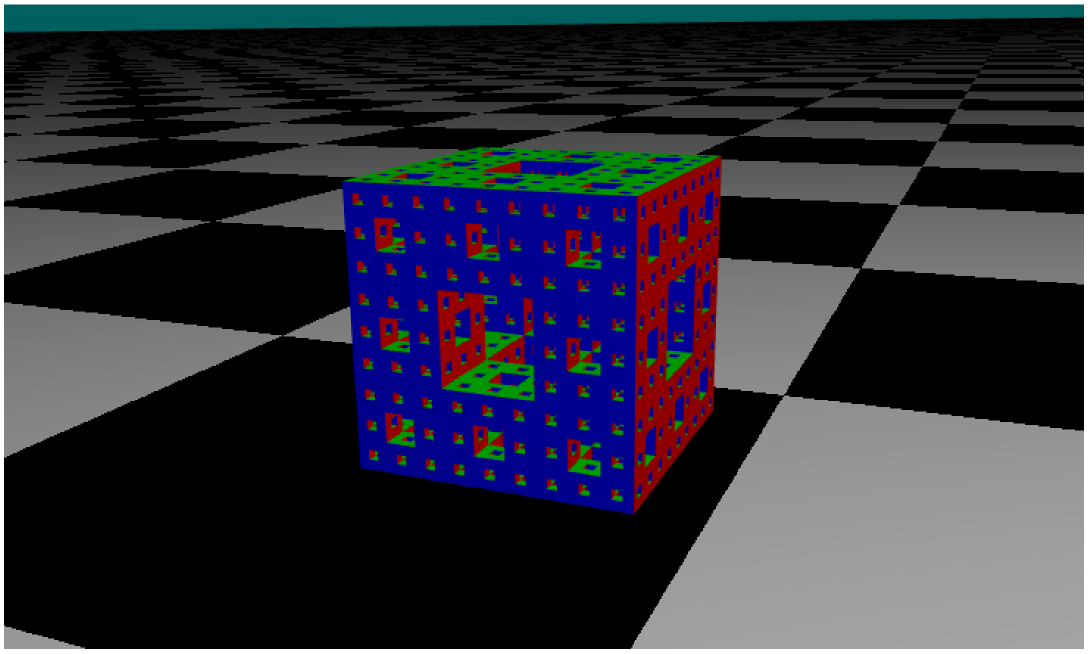


```
//add in the checker pattern
bool white = true;
if(mod(fragcoord.x, 10.0) < 5.0) // z axis-aligned strips
    white = false;
if(mod(fragcoord.z, 10.0) < 5.0) // flip color on every other x axis-aligned strip
    white = !white;
vec4 color = vec4(1.0, 1.0, 1.0, 1.0);
if(white)
    color = vec4(0.0, 0.0, 0.0, 1.0);
```

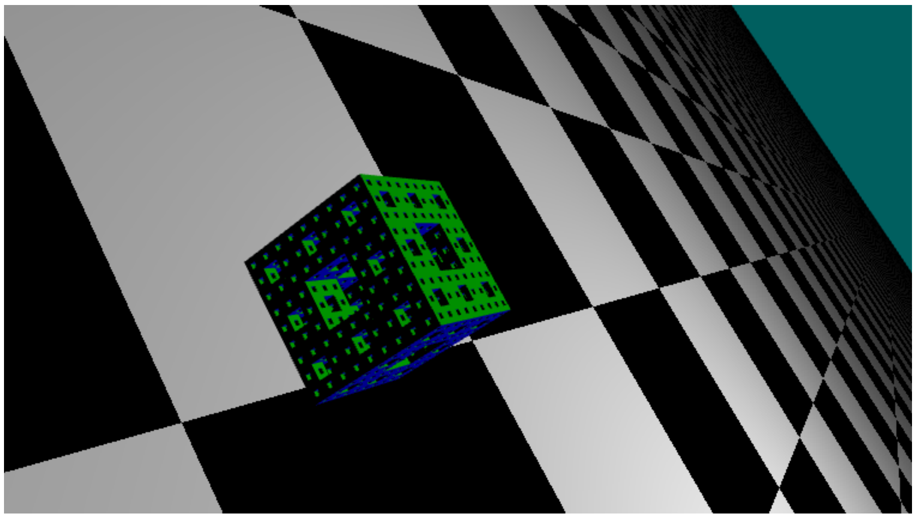
Then, the x and z components of the world coordinate are modded by 10 and then checked to see if they are greater than 5. This is what determines if the fragment is white or black. The floor uses the same code as the sponge to calculate diffuse lighting.

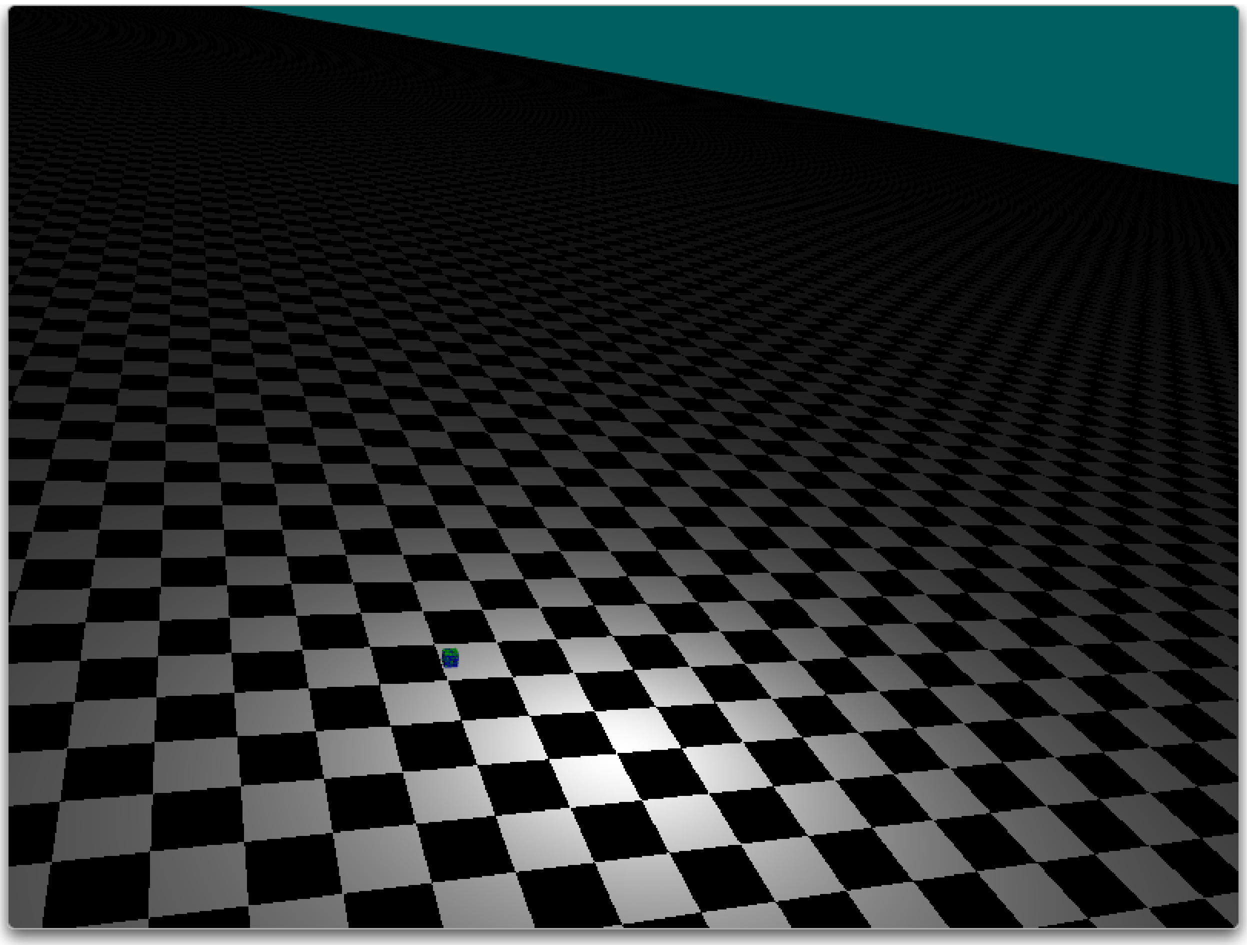
Some pictures:

Different sponge levels:

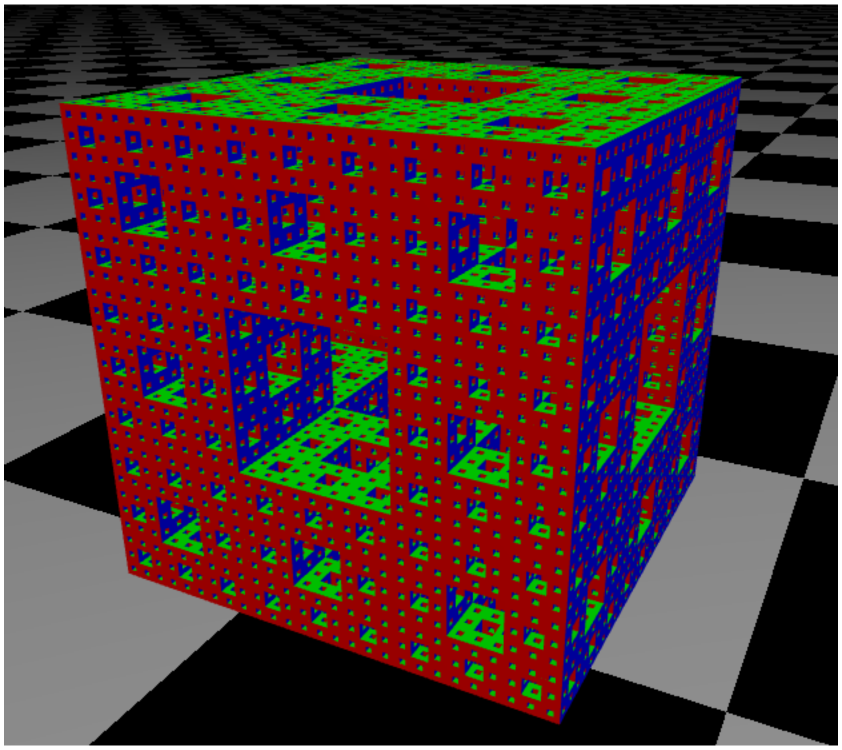
Lighting and weird camera positions:





Bonus fun stuff:

L=5 (takes around a second and a half to calculate on my machine)



Sponge scaled up by 100 (makes the lighting look cool)